CHAPTER

TWELVE

INDETERMINATE APPLICATIVE SYSTEMS

We have already studied two models that, to some extent, express concurrent computation through function composition. Data Flow (Chapter 9) describes concurrency by a process equivalent to the parallel evaluation of function arguments. In Data Flow, a stream of values computed by a primitive is directed to the inputs of its caller. Input streams can be piped together; parallel streams can be processed concurrently. Similarly, the Actors model (Chapter 11) adds concurrency, side effects, and weak fairness to the lambda calculus. Sending a message to an actor resembles calling a function. The primary source of parallelism in Actors is actors that, on receiving one message, send several messages. This roughly corresponds to functions that evaluate several arguments in a single call.

In some sense, Data Flow and Actors start with unusual ideas (graphical and object-oriented computation) and converge toward classical functional systems. This section discusses Indeterminate Applicative Programming (IAP), a set of ideas centered on extending classical functional environments (like pure Lisp and side-effect-free Scheme) into distributable systems. IAP has mechanisms for constructing infinite objects, indeterminate selection, task generation, and communication.

```
(1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  ...
                 The positive integers.

(2  3  5  7  11  13  17  19  23  29  31  37  41  ...
                    The primes.

((1  2  3  4  5  6  7  8  9  10  11  12  13  14  ...
 (2  4  6  8  10  12  14  16  18  20  22  24  26  ...
 (3  6  9  12  15  18  21  24  27  30  33  36  39  ...
                          ⋮
                  A stream of streams.

(⊥  ⊥  ⊥  4  ⊥  6  7  ⊥  9  10  11  12  ⊥  14  15  16  ...
A stream, some of whose elements are undefined.
```

Figure 12-1 Typical streams.

Infinite Objects

Several inventions led to the development of Lisp-based distributable systems. Clearly, one of the most important was Lisp itself [McCarthy 65]. Another, less well known contribution was Peter Landin's description of streams [Landin 65]. *Streams* are possibly infinite sequences of values. Sequences are a familiar idea in computer science—for example, linked lists are an implementation of sequences. Conceptually, a stream is a sequence with no last element. Streams are typically arguments to functions; those functions act on each element of the stream, producing another stream as output. In a Lisp-like system, the constituent values of a stream can be atomic symbols, finite lists, or themselves streams. One value that might occur in a stream is bottom ($\perp$), the undefined value.* Figure 12-1 shows several typical streams. We use Lisp "parenthesis notation" to show values, with unbalanced parentheses and ellipses suggesting infinite streams.

If the arguments to functions are not simply values but streams, then function composition produces a "graphical" description of computation. For example, the functional expression

$$f(g(x,y),h(y,z))$$

can be thought of as a three-node network, where x, y, and z are the input sources and the value produced is the output sink. Figure 12-2 illustrates this relationship. This view of functional programming is similar to Data Flow.†

* $\perp$ (undefined) should be thought of as the result of performing a nonterminating computation such as finding $f(5)$, when the definition of f is $f(x) \equiv 1 + f(x+1)$.

† Perhaps the first to notice the equivalence of function composition and graphical computation over infinite sequences was Gilles Kahn [Kahn 74]. He later extended and amplified this work with David MacQueen [Kahn 77].

Figure 12-2 $f(g(x, y), h(y, z))$.

Any reasonable implementation of infinite sequences makes certain obvious choices. For example, the same function should return the first element of both finite and infinite sequences. However, there are some obvious difficulties with the implementation of infinite objects. For example, given that you are going to have an infinite object, where do you store it? The resolution of this problem lies in not creating "too much" of the infinite object. More succinctly, the optimal strategy is to generate the parts of the infinite object as they are used. This *delayed evaluation* requires modifying the underlying programming system. Since Lisp (like virtually every common programming language except the call-by-name feature of Algol 60) evaluates a function's arguments before evaluating its body,* a function whose argument constructs an infinite object would never terminate.

There are two classes of remedies to this problem. The simpler way (from a programming standpoint) is to make the evaluation scheme be normal-order evaluation. *Normal-order evaluation* evaluates functions from the "outside-in." It substitutes the (unevaluated) arguments to the function for the corresponding bound variables in the function body, and then evaluates the function body. Hence, if a particular function parameter is never used, the corresponding actual parameter is never evaluated.

The opposite of normal-order evaluation is applicative-order evaluation. In *applicative-order evaluation*, the function's arguments are first evaluated and the resulting values are substituted into the body of the function. The body of the function is then evaluated. Lisp expr's and Scheme use applicative-order evaluation.

Some expressions terminate when evaluated in normal order, but not in applicative order. In particular, if the evaluation of an argument to a function produces an infinite object, then applicative-order evaluation creates (or at least tries to create) the entire infinite object. On the other hand, normal-

* At least expr's in Lisp evaluate their arguments in this way.

order evaluation builds only as much of the object as the computation uses.* Infinite data objects always remain finitely described. However, normal-order evaluation has its drawbacks. If a parameter's value is accessed several times during the evaluation of a function's body, then that parameter is reevaluated each time. A system can waste energy repeatedly evaluating the arguments to functions.† Jean Vuillemin and Christopher Wadsworth discovered resolutions of the problem of multiple argument evaluation in normal-order systems in the early seventies [Vuillemin 74; Wadsworth 71]. We call their idea call-by-need. In a *call-by-need* system, evaluation is done in normal order. However, after computing the value of a parameter, the system remembers that value and does not evaluate that parameter again. Three promising (and similar) approaches have been suggested for embedding call-by-need in a Lisp-like system: Gilles Kahn and David MacQueen's "networks of parallel processes" [Kahn 74; Kahn 77]; Peter Henderson and James Morris's "lazy evaluator" [Henderson 76]; and Daniel Friedman and David Wise's "suspending cons" [Friedman 76].

Kahn and MacQueen recognized that collections of Lisp-like functions over streams are networks that perform a Data Flow-like token passing. Making this token passing demand-driven results in call-by-need. Henderson and Morris modified a Lisp-like interpreter to produce call-by-need. Friedman and Wise found a simple and direct way to have infinite objects in Lisp-like systems, suspending cons.

Suspending Cons

In Lisp, the primary data structure is the list. Lists are constructed with the dyadic function cons. Letting nil denote the empty list ( ), the expression (cons 1 nil) evaluates to the list whose only element is 1, that is, (1). Similarly, the expression

```
(cons (cons 5 nil) (cons 3 (cons 8 nil)))
```

evaluates to the list

```
((5) 3 8)
```

* The ability of normal-order evaluation to compute some functions that applicative-order cannot is not limited to infinite objects; it extends to infinite computations. For example, if we define $f(x) \equiv f(x+1)$ and $g(y,z) \equiv \textbf{if } y = 0 \textbf{ then } 1 \textbf{ else } z$, then $g(0, f(5))$ is defined for normal-order evaluation but not for applicative-order evaluation. Normal-order evaluation of this expression yields 1.

† In a pure system (one without side effects) normal-order evaluation is semantically equivalent to performing call-by-name evaluation and applicative-order evaluation is equivalent to call-by-value. Wand [Wand 80] presents a good development of the theory of evaluation and the differences between call-by-name and call-by-value.

We use the traditional Cambridge prefix notation for describing Lisp functions. In Cambridge prefix, the function name goes inside the parentheses with its arguments (conveniently forming a list). Thus, (f x y) represents $f(x, y)$.

Corresponding to function cons there are functions car (first) and cdr (rest) for retrieving the parts of a cons. By definition [McCarthy 63]

$$(\mathsf{car}\ (\mathsf{cons}\ \mathsf{x}\ \mathsf{y})) = \mathsf{x}$$

and

$$(\mathsf{cdr}\ (\mathsf{cons}\ \mathsf{x}\ \mathsf{y})) = \mathsf{y}$$

Traditionally, cons is implemented by allocating a new storage record (a cell) for each call to cons. This record has two fields. A pointer to the first argument (car) of the cons is stored in one field and a pointer to the second argument (cdr) is stored in the other. The basic Lisp implementation recognizes two major varieties of data objects: cons cells (lists) and atoms (tokens).* The key idea in suspending cons involves making one small change to a pure (side-effect-free) Lisp system. With *suspending cons*, the constructor function, cons, does not evaluate its arguments. Instead, a call to cons builds a cell containing two *suspensions*. When a probing function car (or cdr) is called on a cell that contains a suspension, it *forces* that suspension to become a *manifest* ("real") value. That is, car (cdr) evaluates the first (second) argument of the cons that created that cell, *in the environment of the original call to* cons. The result of this evaluation is then stored back into the car (cdr) field of the cons cell in place of the suspension. It is marked as manifest. The next time the car (or cdr) of that cell is desired, the system recognizes that there is a manifest value in the cell (not a suspension), and returns it without further computation. According to Friedman and Wise [Friedman 78a, p. 931]:

> A "suspension" is a temporary structure planted within the field of a record when it is created instead of the value which rightfully should be there. It contains information sufficient to derive that value at any time it is necessary to the course of the computation. In terms of Lisp, sufficient information is the "form" which specifies the value of the field and the "environment" which retains all bindings necessary to evaluate that form at any time in the future.

Cons builds data structures, and data structures are traditionally understood with diagrams. We diagram cons cells as rectangular blocks, with halves for the car and cdr fields. We draw suspensions as "clouds." Therefore, evaluating

* This is just a sketch of a few of the ideas in Lisp. The original Lisp documentation [McCarthy 65] is still a good source for the theory and practice of Lisp. Many books on Lisp have been written in the last few years; Allen [Allen 78] describes Lisp for the system builder, while Winston [Winston 81] provides an introduction to programming in Lisp, particularly for Artificial Intelligence applications.

Figure 12-3 ((lambda (x y)(cons x y)) (+ 3 4) (cons 7 nil)).

Figure 12-4 After evaluating the cdr.

```
((lambda (x y)(cons x y)) (+ 3 4) (cons 7 nil))
```

initially produces the structure illustrated in Figure 12-3. To find the cdr of this structure, we force the cdr-field suspension to become a value. This forcing progresses only as far as the next call to cons. The resulting structure is shown in Figure 12-4. Using suspending cons, we can easily produce infinite structures.

It is important that the reader accept the premise that infinite objects really exist and can be represented in a finite computer system. One can treat an infinite object just like any other object. However, one must be wary of using infinite objects in algorithms that examine the entirety of objects, such as printing or counting the length of an infinite list.

Infinite objects are the children of recursion. We can define infinite objects in two ways: by functional recursion or by data recursion. Recursive function definition should be familiar to most readers. For example, the traditional (recursive) definition of factorial is

```
(factorial n) ≡ (if (= n 0)
                    1
                    (× n (factorial (− n 1))))
```

This is a "good" recursive definition in that (for nonnegative integers n) it always terminates with a well-defined answer. With a suspending cons, we can define function successors as

```
(successors n) ≡ (cons n (successors (+ n 1)))
```

This function produces the infinite list of integers, starting with its argument. That is, the successors of a number is that number consed onto the list that is the successors of one more than it. Thus, (successors 4) is the infinite list (4 5 6 7 8 9 . . . .

Data recursion [Ashcroft 77] may be unfamiliar. With data recursion we define an object in terms of itself. The keyword that indicates self-definition is letrec. Thus, the expression

```
(letrec (suc4 ← (cons 4 (mapcar add1 suc4)))
    <body>)
```

binds to suc4 the stream of integers starting at 4 in the evaluation of <body>.*

Letrec contrasts with the usual use of self-mention in variable assignment. In most languages, x := x + 1 means that the value of x is to be increased by one. With letrec, the equality implies current substitutivity (as in classical mathematics). That is, the structure being built is used in determining the structure being built. This is useful only if the building process examines only those parts that have already been built. Letrec stands for "let, recursively." We use the simple command let when the definition is not recursive.

We illustrate this idea with a program for generating *all* the primes. Our algorithm is the traditional sieve of Eratosthenes; we repeatedly select the smallest remaining number and discard its multiples.

```
(sieve ints) ≡ (cons (car ints)
                     (sieve (removemults (car ints)
                                         (cdr ints))))
```

* This example uses functions add1 and mapcar. The expression (add1 x) returns one more than x. Mapcar takes a function and a list of arguments and builds a list of the results of applying that function to each element of the given list. Hence, if m is the list (1 4 7 10), (mapcar add1 m) is the list (2 5 8 11).

Function removemults takes a number and a list and removes all multiples of its first argument from its second argument.*

```
(removemults n l) ≡
    (cond ((evenly-divides (car l) n) (removemults n (cdr l)))
          (t (cons (car l) (removemults n (cdr l)))))
```

The object that is the stream of all the primes, in increasing order, is the value of

```
(sieve (successors 2))
```

Most recursive Lisp functions have both a base clause (or clauses) and a recursive clause (or clauses). Functions that work with infinite objects usually have only recursive clauses. That is, such programs do not check to see if the problem has been reduced to the empty list because it is never reduced to the empty list. Correspondingly, our sieve and removemults functions only work on infinite lists.

Two-three-five Dijkstra presents the following example, attributing it to Hamming [Dijkstra 76]: Generate, in increasing order, the sequence of all numbers of the form $2^i \cdot 3^j \cdot 5^k$, for natural numbers i, j and k. That is, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, ... Dijkstra requests only the first 100 elements of the sequence. With suspending cons, we can create the entire sequence [Friedman 78b].

We define functions scalar-product and join. The expression (scalar-product s v) is a list of the product of s and each element of v; join merges a pair of sorted streams into a single, sorted stream.

```
(scalar-product i m) ≡
    (cons (times i (car m))
          (scalar-product i (cdr m)))

(join l m) ≡
    (cond ((< (car l) (car m))
           (cons (car l) (join (cdr l) m)))
          ((> (car l) (car m))
           (cons (car m) (join l (cdr m))))
          (t (cons (car l) (join (cdr l) (cdr m)))))
```

* Cond is the Lisp conditional function. The expression (cond (b_1 e_1) (b_2 e_2) ... (b_k e_k)) is equivalent to the conditional expression **if** b_1 **then** e_1 **else if** b_2 **then** e_2 **else** ... **if** b_k **then** e_k **else** nil. That is, cond successively evaluates the b_i until one is true and then returns the value of the corresponding e_i. Cond treats any value that is not nil as true. In particular, the atom t evaluates to itself and is conventionally used for true.

The Hamming sequence is therefore

```
(letrec (comps ← (cons 1
                       (join (scalar-product 2 comps)
                             (join (scalar-product 3 comps)
                                   (scalar-product 5 comps)))))
   comps)
```

Flip-flop Landin observed that once a system includes infinite objects, it is natural to treat a sequential file as an infinite object [Landin 65]. Similarly, we can think of a terminal session as creating an infinite file. Each keystroke adds another element to the terminal stream. This stream has no last element, though all typing after certain sequences may be ignored. Friedman and Wise [Friedman 77] pursued this theme by showing how to write a rudimentary editor as a function of a sequential file stream and a sequential command stream. Their editor produced a sequential file output stream and a stream of responses to the user. In this section we explore the same idea with respect to hardware.

Modeling some situations requires ensuring that the first element of a stream exists before one attempts to examine the rest of the stream [Landin 65]. For example, it is unreasonable to expect to be able to access the character after the next character the user types. For that reason, Friedman and Wise introduce the primitive strictify [Friedman 79]. Strictify is a function of two arguments. It evaluates its first argument and returns the value of its second.

We usually present an example of a simple, state-possessing object (like a register) at this place in the discussion of a model. What does a statelike object look like in a side-effect-free system? In this section, we present a model of a reset/set flip-flop, adapted from a paper by Johnson [Johnson 84].

A *reset/set flip-flop* (RSFF) has two input lines and two output lines. We label the inputs R (reset) and S (set). Figure 12-5 shows the circuit diagram for an RSFF. The RSFF retains its state as long as the inputs are both high. A *pulse* is a sequence of 0s. A pulse on the S line sets the flip-flop (makes Q_{hi}=1 and Q_{lo}=0) and a pulse on the R line resets the flip-flop (makes Q_{hi}=0 and Q_{lo}=1). A *spike* (a single 0) on either line makes the flip-flop unstable. The program for the RSFF is as follows:

```
(NAND l r) ≡ (map2car nand l r)

(nand x y) ≡
      (cond ((= x 0) 1)
            ((= y 0) 1)
            (t 0))

(two_list x y) ≡ (cons x (cons y nil))
```

Figure 12-5 Reset/set flip-flop.

```
(RSFF R S) ≡
     (letrec ($Q_{hi}$ ← (cons 1 (NAND R $Q_{lo}$)))
             ($Q_{lo}$ ← (cons 0 (NAND S $Q_{hi}$)))
        (two_list $Q_{hi}$ $Q_{lo}$))
```

where map2car maps a two-argument function down two lists.

We include the 1 and 0 in the definition of the RSFF to initialize the output. The output of this function is a list of two streams. The first stream is the behavior of Q_{hi}, and the second is the behavior of Q_{lo}.

Indeterminacy

Coordinated computing systems are naturally indeterminate. Purely functional systems (such as Lisp extended by suspending cons) are determinate. That is, every program always returns the same result. Several mechanisms for the addition of indeterminacy to functional systems have been proposed. We discuss three such extensions, two for purely functional systems and the other for suspending cons.

Merge One way of extending a functional system is to provide additional primitive functions. In Chapter 7 we described McCarthy's amb function. This dyadic function indeterminately chooses between its arguments, rejecting undefined arguments in favor of defined ones. Another function for indeterminate processing is merge. Merge takes a pair of input streams and produces an output stream of interleaved elements from both input streams. This is much like taking two decks of cards and shuffling them with a standard shuffle: the two decks are interleaved, but the cards of each deck retain their original ordering.* For example, the merge of the streams (1, 2, 3, 4, ... and (A, B, C, D, ... might be the stream (A, B, 1, 2, 3, C, 4, 5, ... But it also might be the stream (1, 2, 3,

* We presented an example of an indeterminate merge in the our discussion of Data Flow (Section 9-2).

4, ..., 252, A, 253, ..., 319, B, 320, ..., 461, C, ..., or even the stream (1, 2, 3, ... Different definitions of merge take different approaches to streams that include undefined elements. MacQueen's merge [MacQueen 79] specifies that if a stream has an undefined element, then that element and all future elements from that stream are ignored. Turner's merge [Turner 80] passes undefined elements on to the output. We can characterize the difference operationally by thinking of the MacQueen merge as examining the items in its list before passing them to the output and the Turner merge as passing items without examination. An n-element merge can easily be built from two-element merges. Exercise 12-4 asks for generalized merge functions.

Frons Friedman and Wise adopt a different approach to introducing indeterminacy to functional systems. Their invention is frons, an indeterminate constructor. Cons constructs lists whose element order is determined at "construction time." Thus,

```
(cons 1 (cons 2 (cons 3 nil)))
```

produces the list (1 2 3). The car of that list is 1 and will always be 1.

Frons also builds listlike objects, in that car and cdr can be used to extract pieces of the result. However, unlike lists that have been created with cons, the order of the elements of a frons list is not determined until the list is probed by car or cdr. For example, let m be the result of (frons 1 (frons 2 (frons 3 nil))). The first call for car of m may yield 1, 2, or 3. If (car m) evaluates to 2 at some time, then it will always be 2. If (car m) is 2, then (car (cdr m)) will be either 1 or 3. In general, car pulls a convergent (not $\perp$) element from a frons list; cdr forces a selection of the first element and forms the remaining elements into a frons list. Forcing the evaluation of the entirety of a frons-list produces some permutation of the original elements. This permanence of "found values" is engineered by changing the frons cell into a cons cell when the value of the car is discovered and placing that value in the car of the cons cell.

Frons never selects a divergent element before a convergent one. In this respect, it resembles merge and amb. Thus, the expression

```
(car (frons (cond ((< x 0) ⊥)
                  (t 1))
            (frons (cond ((> x 0) ⊥)
                         (t 1))
                   (frons ⊥ nil))))
```

always evaluates to 1, no matter what the value of x (provided, of course, that x has a numeric value).

Determining (in general) whether an element of a frons list will converge is formally undecidable. In implementations of frons, elements are selected by distributing processing resources to successive suspensions until one converges. This (almost) orders the elements of a frons list by their relative computation cost.

Figure 12-6 (frons (cons 3 x) (frons 5 (frons (cons 2 y) nil))).

Frons, like suspending cons, does not evaluate its arguments. Instead, it builds suspensions. Hence, one can easily build infinite frons structures. For example, the definition

$$(\mathsf{frsuccessors\ i}) \equiv (\mathsf{frons\ i\ (frsuccessors\ (+\ i\ 1))})$$

implies that (car (frsuccessors 1)) is going to be a positive integer, though there is no way to tell which one. (IAP therefore supports unbounded indeterminacy.) If n is bound to (frsuccessors 1), and we take (car n), we might get the value 21356. The system decides this once for each frons cell; the next time the (car n) is evaluated, it is still 21356. The value of (car (cdr n)) could be any positive integer except 21356.

In our diagrams, we illustrate frons constructions as wavy boxes. Taking the car of a list of such wavy boxes straightens one of these boxes and moves it to the front of the list. To illustrate this idea, we let l be the list

(frons (cons 3 x) (frons 5 (frons (cons 2 y) nil)))

Figure 12-6 shows the box diagram of l. Of course, the frons cells in the cdr of the cell are originally suspensions; we have drawn the entire frons structure to illustrate the evaluation process.

Taking the car of such a list resembles polling several input lines: we are interested in whichever computation converges first. Let us imagine that it is the second element, 5. This causes a promotion of that value to the head of the list, the solidifying of its wavy box into a rectangular cons cell, and the rearrangement of the rest of the list. Figure 12-7 shows the resulting state.

Sharing of sublists complicates promotion algorithms. For an example of the difficulties involved, let k be the list (frons 1 (frons 2 nil)), and j the list created by (frons 9 k). Figure 12-8 shows this situation. Now, imagine that we evaluate the (car j) and it turns out to be 2. A naive promotion algorithm would rearrange the elements of the frons list so that 2 was the first element, as shown in Figure 12-9. To which element should k now point? By the definition of frons, k must be either the list (1 2) or the list (2 1). It can neither skip one of these

Figure 12-7 After taking the car of l.

elements nor include the spurious 9. One resolution of this problem comes from copying. Figure 12-10 shows a correct promotion. A correct promotion algorithm appears in the original frons paper [Friedman 79].

Multiprocessing The reader may be curious as to why we include IAP in a book on coordinated computing. After all, many of the conventional aspects of distributed computing, such as explicit processes, communication primitives, and distance, are not mentioned by this system. The importance of IAP is that each suspension created by a cons or frons operation is an independent task. A multiprocess system can devote a process to expanding each suspension. When a processor is idle, it can look for a suspension in need of expanding. However, these processes must not get too far ahead of the main computation. We would be unhappy with the diligent process that expanded, say, the first 7 million primes when the program uses only the first seven.*

Figure 12-8 Sharing in frons lists.

* Friedman and Wise have identified an algorithm for distributing processing energy among competing suspensions. Their algorithm gives greater energy to computations closer to the main derivation [Friedman 79].

Figure 12-9 Incorrect promotion scheme.

Figure 12-10 Correct promotion.

How can multiple active agents share a lock-free space? To solve this problem, Friedman and Wise invented the operator sting. Sting is a conditional store operation. In the multiprocess model of an IAP system, processes strive to evaluate suspensions, turning suspensions into manifest structures. When an active processor finds the value of a suspension, it "stings" the pointer cell of that suspension with that value. If the cell has already been stung, then nothing happens. Only if that cell has not been stung before does the system insert the value in the cell and mark the cell as stung. The stinging process is not informed if its sting succeeded. If several processors are working on resolving the same suspension (as in the car of a frons list), the first one to finish has a successful sting. Subsequent stings are ignored. Sting combines aspects of send-and-forget and test-and-set instructions. Since IAP is side-effect-free at the processor level, cells progress exactly once from frons cells to cons cells.

The principle underlying the cons and frons functions is turning processes into objects. Johnson and Kohlstaedt describe this philosophy in analogy to Lisp [Johnson 81, p. 4]:

> Returning to the Lisp analogy, we note that one intent of DSI [an implementation of IAP ideas] is to do with *process* what Lisp does with *data*. Lisp "the list processor" is a primeval data management system; it "factors out" some of the complexity of data manipulation by reducing structure to an elemental form—the binary list cell. Similarly, we seek a "lowest common denominator" for the notion of process. The focal point of our discussion is the *suspension*, a kinetic counterpart to the inert list cell. Where, in our view, data is fixed and immutable, suspensions evolve in the presence of processing.

The idea behind frons and cons is to relieve the programmer from worrying about the details of scheduling. A program using frons appears to the programmer like a theater with a single narrow exit door. At some point people (tasks) enter the system. When they leave, it is single file, one at a time. The programmer can remain oblivious to the internal organization of the departure. The exiting patrons can leave in an orderly progression or someone can yell "fire," causing chaos in the theater. The door emits only the next patron. The user of a frons list gets only the next element of the list. The user need not worry how much jockeying for position occurred to get that list in order. The list is in order when needed.

Scheduling

Scheduling freedom is important for applications involving not only operating systems concerns, but also search. To show the scheduling power of frons and cons, we present three examples—a simple search, a selection among algorithms, and a small operating systems scheduler.

Good sequences Dijkstra [Dijkstra 72b] defines a *good sequence* as a sequence of 1s, 2s, and 3s that is not of the form $xyyz$, for any subsequences x, y, and z (y not null). Thus, the sequence

$$1\ 2\ 3\ 2\ 1\ 3\ 2$$

is a good sequence, while the sequences

$$1\ \underline{2\ 3}\ \underline{2\ 3}\ 1$$

and

$$1\ 3\ 1\ 2\ \underline{3\ 2\ 1}\ \underline{3\ 2\ 1}\ 2\ 3\ 2\ 1$$

are not.

The good sequences problem asks for a program that, given a length of sequence desired, produces a good sequence of that length. One way of programming the problem is depth-first, recursive search. This algorithm tries extending the current candidate in all possible ways until some sequence reaches the desired length. If all fail, the search backtracks.

Our program for the good sequences problem retains the state of all partial solutions. That way, if we want a different or longer solution, we can extend an earlier partial solution. This avoids starting the search process over from the beginning.

We assume the existence of function good?. The value of (good? x) is true if x is a list whose elements form a good sequence. A recursive program, step, that takes a good sequence, sequence, and extends it in all possible ways with how-many-more additional elements is as follows:

```
(step sequence how-many-more) ≡
    (cond ((good? sequence)
            (cond ((= how-many-more 0) (cons sequence nil)
                  (t (stepper sequence (sub1 how-many-more) 3))))
          (t (frons ⊥ nil)))

(stepper sequence how-many-more k) ≡
    (cond ((= k 0) nil)
          (t (frappend (step (cons k sequence) how-many-more)
                       (stepper sequence how-many-more (sub1 k)))))

(frappend l m) ≡
    (cond ((= l nil) m)
          (t (frons (car l) (frappend (cdr l) m))))
```

A program that needs good sequences of length 5 might bind to a variable x the value of (step nil 5). The value of (car x) is a good sequence. If that particular good sequence proved unappealing, the (car (cdr x)) would also be a good sequence (as would the rest of the elements of x). The first sequence in x would be the first good sequence to be found. If later we decide that we really need good sequences of size 7, we could evaluate (frapcar (lambda (z) (step z 2)) x).*

Algorithmic selection Problems often have several alternative algorithmic solutions. We cannot always decide beforehand which is best (and which are impractical). For example, often many different algorithms solve the same class of numeric problems, but do better over different ranges of input. One algorithm might be good for large values or inexact answers and yet be divergent near zero. It may be difficult to choose, a priori, the best algorithm for a given problem.

* Frapcar is the frons analogue of mapcar: (frapcar f l) ≡ (cond ((= l nil) nil) (t (frons (f (car l)) (frapcar f (cdr l))))). The 2 in the function call is the extension of five-element sequences by two to be seven-element sequences.

Frons relieves the programmer from having to make that choice. The programmer can frons together a list of expressions, where each expression represents one possible algorithm. The car of that list would be the answer provided by the first algorithm to converge.

For example, a system to test numbers for primeness might define two functions, say fact and prob-prime. Function fact would seek factors for its argument; function prob-prime would perform a probabilistic test for primeness. Each would return after satisfying its test (and not return if it failed to satisfy). The function (lambda (x) (car (frons (fact x) (frons (prob-prime x) nil)))) would schedule the quicker response.

Terminal controller In "Circuits and Systems," Johnson addresses the problem of a terminal controller [Johnson 84]. He describes a full-duplex message system for two users, u_1 and u_2. Normally, the system echoes the input from each user's keyboard (k_1 or k_2) on that user's screen (s_1 or s_2). However, when a user executes a send command, the input echoes on the other user's screen. The architecture of the message system is shown in Figure 12-11. The MSG function produces an output of two streams, one for each terminal, from an input of two streams, one from each keyboard.†

```
(MSG k1 k2) ≡
    (letrec (r1  ← (route k1 m2))
            (r2  ← (route k2 m1))
            (s1  ← (car r1))
            (m1 ← (car (cdr r1)))
            (s2  ← (car r2))
            (m2 ← (car (cdr r2)))
       (two_list s1 s2))


(route k min) ≡
    (let (w ← (wire (cons (two_list (quote ON) (quote #))
                          (select k))))
      (let (mout ← (car w))
           (dplx  ← (car (cdr w)))
         (two_list (merge_input dplx min) mout)))
```

† Sometimes we want a convenient way to refer to a list (or atom) in a program without binding or building it from scratch. For that we use function quote. Quote "quotes" its argument. That is, quote does not evaluate its argument. Instead, its value is the literal structure of its argument. The value of (quote #) is #; the value of (quote (1 4 7 10)) is the list (1 4 7 10). For example, evaluating (mapcar add1 (quote (1 4 7 10))) yields the list (2 5 8 11). Since numbers and functions evaluate to themselves, they need not be quoted.

```
(select k) ≡
     (let (ka ← (car k))
          (kd ← (cdr k))
        (cond ((= ka (quote SEND))
               (let (kda ← (car kd))
                    (kdd ← (cdr kd))
                  (cons (cons (quote #)
                              (two_list (quote ?) kda))
                        (select kdd))))
              (t (cons (cons ka (quote #))
                       (select kd)))))

(merge_input l r) ≡
     (let (la ← (car l))
          (ld ← (cdr l))
          (ra ← (car r))
          (rd ← (cdr r))
        (car (frons (strictify la (cons la (merge_input r ld)))
                    (frons (strictify ra (cons ra (merge_input l rd)))
                           nil))))

(ones s) ≡
     (cond ((= (car (car s)) (quote #))
            (ones (cdr s)))
           (t (cons (car (car s))
                    (ones (cdr s)))))

(twos s) ≡
     (cond ((= (car (cdr (car s))) (quote #))
            (ones (cdr s)))
           (t (cons (car (cdr (car s)))
                    (twos (cdr s)))))

(wire s) ≡ (two_list (ones s) (twos s))
```

Function route takes keyboard input and message input and writes screen output and message feedback. Figure 12-11 shows the input-output relationships of route. The (quote ON) in route is for stream initialization. Merge_input indeterminately checks both input lines (using frons) and selects input from the first available line.*

* Other examples that use this approach include Henderson's description of an operating system [Henderson 82] and Keller and Lindstrom's functional graph language [Keller 81].

Figure 12-11 The terminal-message system.

Cull

We have already seen several different operators for introducing indeterminacy into applicative systems — amb, two varieties of merge, and frons. There are others. In this section we indulge one of our periodic fantasies and consider the inverse operator to indeterminate merge. This fantasy was prompted by some remarks by Turner [Turner 80].

Indeterminate merge takes two input streams and interleaves them onto a single output stream. Every element in each input stream appears in the output stream (except when merge chooses to ignore the remainder of one stream). We call the inverse operator of merge, split. Split takes a single input stream and produces two output streams such that each element in the input stream appears in exactly one output stream. For example, the expression

```
(let (s ← (split (successors 1))
  (let (a ← (car s))
      (b ← (car (cdr s)))
    <body>))
```

binds both a and b to an ordered subsequence of the positive integers (in <body>). Every integer appears in either a or b, and no integer appears in both a and b.

We can easily obtain this behavior by having split "flip a coin" on each element of the list; "heads" elements going to the first output, and "tails" going to the second. This is unsatisfactory, because we really want a and b's demands to grant them list elements. So we add a further restriction: a and b are to pull elements from the split list as they need them, where need is defined by call-by-need. Hence, if <body> never references a, then the entire list should appear in b; if <body> uses the first 10 elements of a and only the first three elements of b, then the fourteenth element should not appear in either a or b. This restriction precludes the coin flipping solution.

Split is a natural controller for task scheduling. For example, an unbounded producer-consumer buffer of two inputs (producers) and two outputs (consumers) is as follows:

```
(buffer l m) ≡ (split (my_merge l m))

(my_merge l m) ≡
    (let (x ← (frons (strictify (car l) l)
                     (frons (strictify (car m) m) nil)))
      (cons (car (car x))
            (my_merge (cdr (car x))
                      (car (cdr x)))))
```

We use strictify in this example to ensure that the producer's item exists before it is given to the consumer. Otherwise, this buffer would allocate promises of buffer items that might never exist.

One simple way of introducing split would be to posit a new variety of box (as cons and frons are varieties of boxes). This box would have a mark bit that would be set if its contents had already been taken. A system primitive could filter lists of these boxes, discarding those with marks, and marking and passing those without. Such a mark resembles a lock or semaphore.

This alternative is unesthetic. Another variety of box would require modifying the other primitives of the system to reflect its existence. After all, one ought not multiply entities beyond need.

What are the primitive data types of IAP? In addition to atoms, IAP distinguishes frons cells from cons cells and suspensions from manifest values. Each of these is implemented by a mark/unmark bit. When a frons cell becomes a cons cell, a one-bit field in the cell is updated. Similarly, when a suspension becomes manifest, a different one-bit field is changed. For any given cell, there are two manifest bits, one for each of the car and cdr fields. This implies that there can easily be a function existscdr? that checks if the cdr field of a cell contains a manifest (not suspended) value. Using that as our test for the marking bit, we get function cull*

```
(cull l) ≡
      (cond ((= l nil) nil)
              ((**not** (existscdr? l))              -- test if the cdr contains a suspension
               (let (x ← (car l))
                    (y ← (cdr l))
                  (strictify y (cons x (cull y)))))
              (t (cull (cdr l))))
```

We can then write split in terms of cull as

```
(split l) ≡
      (let (m ← (package l))
         (two_list (cull m) (cull m)))

(package l) ≡
      (cond ((= l nil) nil)
              (t (cons (car l) (package (cdr l)))))
```

Package is equivalent to a top-level copy of its argument. Split passes its argument through package to ensure that the entire list is composed of suspensions.

Much like frons, cull plays a restrained havoc with substitutivity. For example, it is usually the case that

$$(\text{cull l}) \neq (\text{cull l})$$

while

$$(\text{cull (package l)}) = (\text{cull (package l)})$$

And, of course,

$$(\text{strictify (print (cull l)) (cull l)}) = \text{nil}$$

Perspective

IAP approaches coordinated computing from a different direction than most of the other systems. In IAP, processes become data; tasks are resolved by free processor resources. This shifts the responsibility for indicating concurrency from the user to the underlying system. Nevertheless, IAP remains a programming system; the tasks themselves are expressed as programs, not problem domain expressions.

* If cull is used in a multiprocessing environment, the code ((not (existscdr? l)) ...) must be executed indivisibly. This avoids the problem of two processes consuming the same value. For similar approaches see Peterossi [Peterossi 81] and Clark and Gregory [Clark 81].

PROBLEMS

12-1 Write a function that produces the infinite list of even integers.
12-2 What is the result of evaluating (primes (successors 1))?
12-3 Define a five-element merge operator in terms of a two-element merge operator.
12-4 Define the merge operator that takes a stream of streams and produces their merge.
12-5 What are the possible values of the following expressions?

```
    (a)    (let (x ← (frons 1 (frons 2 nil)))
             (car x))

    (b)    (letrec (x ← (frons (cond ((= (car x) 3) 3) (t 4))
                               (frons (cond ((= (car x) 4) 3)
                                            (t 4))
                                      nil)))
              (car x))

  † (c)    (letrec (x ← (frons (cond ((= (car x) 3) 3) (t 4))
                               (frons (cond ((= (car x) 4) 4)
                                            (t 3))
                                      nil)))
              (car x))
```

12-6 Give an expression whose value is a stream, each element of which is the next digit of the decimal expansion of e (the base of the natural logarithms). That is, the stream begins (2 7 1 8 2 8 1 8 2 8 4 5 9 . . . . Instead of being an approximation to e whose accuracy is determined at the time the approximation is derived, it is the value of e. The application (by its use), not the generation algorithm, determines the number of significant digits generated. (*Turner*)
† **12-7** Give an expression whose value is a stream that is an infinite good sequence.
12-8 What happens if the 1 and 0 are omitted from the definition of the RSFF?
12-9 What happens if R=1=S at the beginning of the definition of the RSFF? (*Wise*)
12-10 Evaluating y and invoking cons in the definition of cull is expensive. Redefine cull so that these two expressions are not in the atomic portion.
12-11 Write a function, setify, that takes an infinite stream of atoms and returns the set implied by that stream—that is, that stream with multiple elements removed. For example, if list m is

(2 2 4 2 4 6 2 4 6 8 2 4 6 8 10 2 4 6 8 10 12 . . .

then the result of (setify m) is some permutation of the even positive integers.
12-12 Write a function that produces the union of two infinite sets of positive integers. Assume that the sets are represented as sorted lists.

REFERENCES

[**Allen 78**] Allen, J., *The Anatomy of LISP*, McGraw-Hill, New York (1978). Allen describes everything you always wanted to know about building a Lisp system.

[**Ashcroft 77**] Ashcroft, E. A., and W. W. Wadge, "Lucid, a Nonprocedural Language with Iteration," *CACM*, vol. 20, no. 7 (July 1977), pp. 519–526. Lucid is a language that combines axiomatic declarations of programs with ease of verification. The last example in this paper, a prime sieve, uses an infinite streamlike data structure.

[**Clark 81**] Clark, K. L., and S. Gregory, "A Relational Language for Parallel Programming," *ACM Proc. 1981 Conf. Func. Program. Lang. Comput. Archit.*, Portsmith, New Hampshire (October 1981), pp. 171–178. This paper adds a notation to Prolog for parallel programming. Clark and Gregory show how to do merge; since programs in Prolog can be run "both ways" this also yields split.

[**Dijkstra 72b**] Dijkstra, E. W., "Notes on Structured Programming," in O.-J. Dahl, E. W. Dijkstra, and C.A.R. Hoare, *Structured Programming*, Academic Press, London (1972), pp. 1–82. Dijkstra presents the good sequences problem (attributing it to Wirth) on pages 63–66.

[**Dijkstra 76**] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, New Jersey, Englewood Cliffs (1976). Dijkstra describes Hamming's "2-3-5" problem on pages 129–134. He wrote his solution imperatively and with guarded commands.

[**Friedman 76**] Friedman, D. P., and D. S. Wise, "CONS Should Not Evaluate its Arguments," in S. Michaelson, and R. Milner (eds.), *Automata, Languages and Programming*, Edinburgh University Press, Edinburgh (1976), pp. 257–284.

[**Friedman 77**] Friedman, D. P., and D. S. Wise, "Aspects of Applicative Programming for File Systems," *Proc. ACM Conf. Lang. Des. Rel. Softw.*, North Carolina (1977), pp. 41–55.

[**Friedman 78a**] Friedman, D. P., and D. S. Wise, "A Note on Conditional Expressions," *CACM*, vol. 21, no. 11 (November 1978), pp. 931–933.

[**Friedman 78b**] Friedman, D. P., and D. S. Wise, "Unbounded Computational Structures," *Softw. Pract. Exper.*, vol. 8, no. 4 (August 1978), pp. 407–416.

[**Friedman 79**] Friedman, D. P., and D. S. Wise, "An Approach to Fair Applicative Multiprograming," in G. Kahn (ed.), *Semantics of Concurrent Computation*, Lecture Notes in Computer Science 70, Springer-Verlag, New York (1979), pp. 203–225.

[**Friedman 80**] Friedman, D. P., and D. S. Wise, "An Indeterminate Constructor for Applicative Programming," *Conf. Rec. 7th ACM Symp. Princ. Program. Lang.*, Las Vegas, Nevada (January 1980), pp. 245–250.

[**Henderson 76**] Henderson, P., and J. H. Morris, "A Lazy Evaluator," *Conf. Rec. 3d ACM Symp. Princ. Program. Lang.*, Atlanta, Georgia (January 1976), pp. 95–103.

[**Henderson 82**] Henderson, P., "Purely Functional Operating Systems," in J. Darlington, P. Henderson, and D. A. Turner, *Functional Programming and its Applications*, Cambridge University Press, Cambridge (1982), pp. 177–192. Henderson presents several examples of implementing operating systems components in a purely applicative language. A key feature of his design is a nondeterminate merge.

[**Johnson 81**] Johnson, S. D., and A. T. Kohlstaedt, "DSI Program Description," Technical Report 120, Computer Science Department, Indiana University, Bloomington, Indiana (December 1981). Johnson and Kohlstaedt provide a good description of the philosophy behind concurrent computation with suspensions.

[**Johnson 84**] Johnson, S. D., "Circuits and Systems: Implementing Communication with Streams," in M. Ruschitzka, M. Christensen, W. F. Ames, and R. Vichnevetsky, R. (eds.), *Parallel and Large-Scale Computers: Performance, Architecture, Applications*, vol. 2 IMACS Transactions on Scientific Computation, North-Holland, Amsterdam (1984), pp. 311–319. This is the source of the flip-flop program.

[**Kahn 74**] Kahn, G., "The Semantics of a Simple Language for Parallel Programming," in J. L. Rosenfeld (ed.), *Information Processing 74: Proceedings of the IFIP Congress 74*, North Holland, Amsterdam (1974), pp. 471–475. This was one of the earliest papers to point out the notion of demand-driven evaluation.

[**Kahn 77**] Kahn, G., and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," in B. Gilchrist (ed.), *Information Processing 77: Proceedings of the IFIP Congress 77*, North Holland, Amsterdam (1977), pp. 993–998. Kahn and MacQueen develop the ideas of turning every edge in a functional graph into a queue and every argument to functions into a stream.

[**Keller 81**] Keller, R. M., and G. Lindstrom, "Applications of Feedback in Functional Programming," *ACM Proc. 1981 Conf. Func. Program. Lang. Comput. Archit.*, Portsmith, New Hampshire (October 1981), pp. 171–178. Keller and Lindstrom show how to model continuous simulation with feedback loops. This paper is a good introduction to their function graph language, FGL.

[**Landin 65**] Landin, P., "A Correspondence Between ALGOL 60 and Church's Lambda Notation: Part I," *CACM*, vol. 8, no. 2 (February 1965), pp. 89–101. This paper demonstrates how Algol programs can be transcribed into an applicative-order lambda calculus. This was one of the earliest uses of streams.

[**McCarthy 63**] McCarthy, J., "A Basis for a Mathematical Theory of Computation," in P. Braffort, and D. Hirschberg (eds.), *Computer Programming and Formal Systems*, North Holland, Amsterdam (1963), pp. 33–70. McCarthy introduces amb.

[**McCarthy 65**] McCarthy, J., P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin, *Lisp 1.5 Programmer's Manual*, M.I.T. Press, Cambridge, Massachusetts (1965).

[**MacQueen 79**] MacQueen, D. B., "Models for Distributed Computing," Rapport de Recherche 351, Institut de Recherche d'Informatique et d'Automatique, Le Chesnay, France (April 1979). MacQueen surveys three models for distributed computing. His paper includes a nondeterminate merge.

[**Peterossi 81**] Peterossi, A., "An Approach to Communications and Parallelism in Applicative Languages," *International Conference on Automata, Languages, and Programming*, Lecture Notes in Computer Science 107, Springer-Verlag, New York (1981), pp. 432–446. Peterossi proposes a communication structure to make functional programs more efficient. His structure is a shared message queue, with operators to add to the queue, remove an element from the queue so that no other process sees it, and remove an element from the queue so that other processes continue to see it. Thus, his queues can serve to cull lists.

[**Steele 78**] Steele, G. L., Jr., and G. J. Sussman, "The Revised Report on SCHEME, a Dialect of Lisp," Memo 452, Artificial Intelligence Laboratory, M.I.T., Cambridge, Massachusetts (January 1978).

[**Turner 80**] Turner, D. A., personal communication, 1980.

[**Vuillemin 74**] Vuillemin, J., "Correct and Optimal Implementation of Recursion in a Simple Programming Language," *J. Comput. Syst. Sci.*, vol. 9, no. 3 (June 1974), pp. 332–354. Vuillemin introduces call-by-delayed-value, an independently discovered form of call-by-need.

[**Wadsworth 71**] Wadsworth, C., "Semantics and Pragmatics of the Lambda-calculus," Ph.D. dissertation, Oxford University (1971). Wadsworth describes call-by-need.

[**Wand 80**] Wand, M., *Induction, Recursion, and Programming*, North Holland, New York (1980).

[**Winston 81**] Winston, P. H., and B.K.P. Horn, *LISP*, Addison-Wesley, Reading, Massachusetts (1981). This book is an introduction to Lisp. It emphasizes examples from Artificial Intelligence.